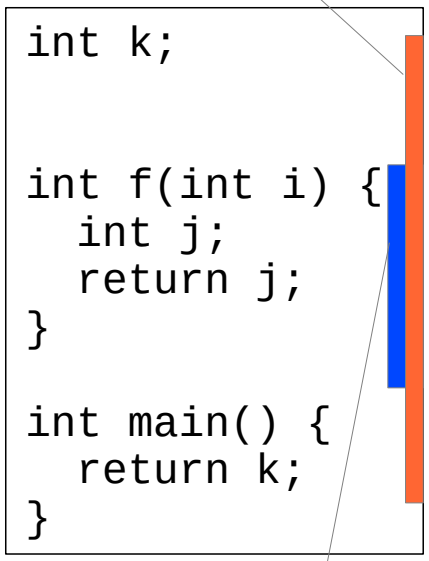
Sichtbarkeit und Lebensdauer von Variablen

Vorkurs C/C++, Olaf Bergmann

Lokale und globale Variablen

- Lokale (automatische) Variablen
  - Nur in umgebendem Block sichtbar (z. B. Funktion)
  - Nach Verlassen des Blocks zerstört
  - Nicht initialisiert
- Globale Variablen
  - Außerhalb von Funktionen deklariert
  - Sichtbar von Deklaration bis zum Ende der Quelldatei
  - Sichtbar in allen geschachtelten Blöcken (z. B. Funktionen)
  - Lebendig bis Programmtermination
  - Globale Variablen sind mit Null initialisie
- Funktionen sind immer global

Sichtbarkeit von k
Lebensdauer von k
k ist mit 0 initialisiert



Sichtbarkeit von j
Lebensdauer von j
j ist nicht initialisiert!

Sichtbarkeit und Lebensdauer

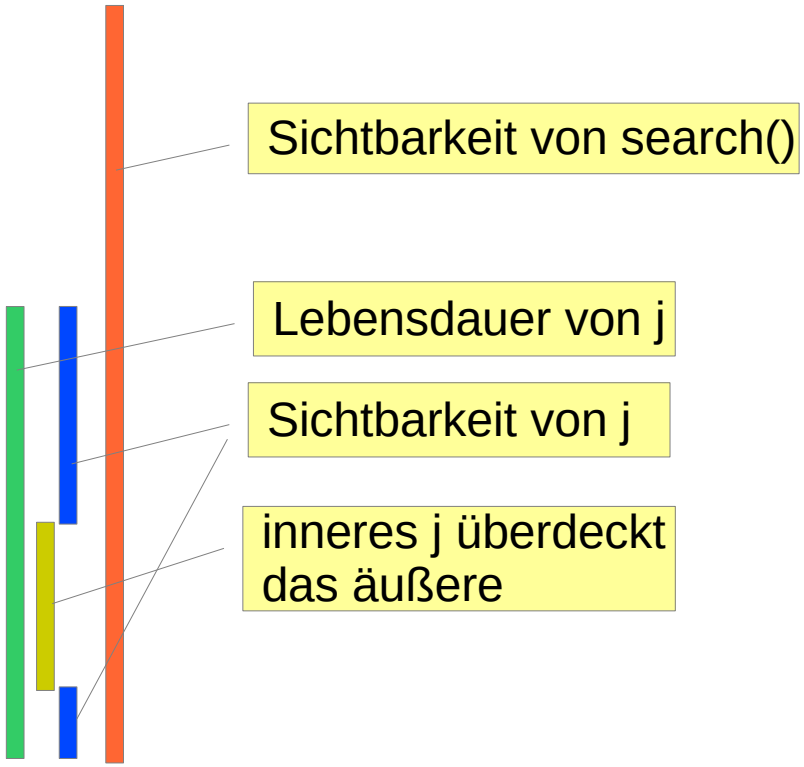
```
const int maxstack = 100;

void search(int i);

int main() {
  search(15);
}


void search(int i) {
  int j = 0;

  while (j != i) {
    ...
    {
       float j;
       ...
    }
  }
}
```



Extern-Deklaration

- Deklariert eine Variable vor ihrer Benutzung
  `extern int size;`
- Nötig, wenn Variable in einer anderen Quelldatei definiert
- Macht Typ der Variable bekannt
  - Compiler legt keinen Speicherplatz an
  - Keine Initialisierung
  - Global oder lokal
- Variablendefinition wie gewohnt
  `int size = 100;`
  - Nur eine Definition erlaubt
  - Veliebig viele `extern`-Deklarationen
- Funktionsdeklarationen sind immer extern (`extern` redundant)

Statische Variablen

- Globale statische Variablen
  - sichtbar nur bis Ende der Datei
  - In anderen Dateien nicht sichtbar/importierbar
  - Ziel: Modularisierung; Vermeidung von Namenskonflikten

```
const int maxstack = 100;
static int stack[maxstack], sptr = 0;
```

- Lokale statische Variablen
  - Sichtbar wie automatische lokale Variablen
  - Bestehen auch nach Verlassen des Blocks
  - Mit Null initialisiert

```
for_each(s.begin(), s.end(),
  [](char c) {
    static size_t column = 0;
    if (++column >= 76) {
      cout << endl;
      column = 0;
    }
  }
);
```

Zusammenfassung: Sichtbarkeit und Lebensdauer

	Speicher-klasse	Sichtbarkeit	Lebensdauer
global	`extern`	unbegrenzt	unbegrenzt
	`static`	Datei	unbegrenzt
lokal	`auto`	Block	bis Blockende
	`static`	Block	unbegrenzt

häufig genutzte Variablen optional in Prozessorregistern

Lebensdauer von Objekten

grundsätzlich wie andere Variablen:

```
class Objekt {
  ...
};

Objekt f(void) {
   Objekt k;
   return k;
}

int main() {
   Objekt x;
   x = f();
}
```

Erzeugt eine Kopie von k

x existiert bis zum Ende von main()

zuweisen der Kopie von k

Speicherverwaltung

- In C: Bibliotheksfunktionen malloc()/free()
  - Liefert void *, keine Typsicherheit (vgl. Alignment)
- C++: Operatoren new/delete
  - Vor allem für Objekte und komplexe Datentypen

```
int *p = new int;
...

 /* Aufräumen nicht vergessen,
    sonst Speicherleck! → valgrind */
delete p;
```

Speicher eines Objekts freigeben

Arrays

```
int *p = new int[expr];
...
delete[] p;
```

Speicher eines Arrays von Objekten freigeben

Varianten von new

- `bad_alloc`: Speicheranfrage nicht erfüllbar

```
int *p = new int[100000000000];
std::cout << p << std::endl;

→ terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

- alternativ:

```
#include <new>

int *p = new (std::nothrow) int[100000000000];
std::cout << p << std::endl;

→ 0
```

new und delete überladen

- C++: Operatoren new/delete

```
void *operator new(size_t s);
void operator delete(void *p);
```

- Eigene Definition durch Überladen

```
#include <cstdlib>  /* malloc/free */

void *operator new(size_t s) {
  void *p = malloc(s);
  printf("neu: %p (%zu Bytes)\n", p, s);
  return p;
}

void operator delete(void *p) {
  printf("entf: %p\n", p);
  free(p);
}
```

Achtung:
`throw(std::bad_alloc)`

Generell gilt sowieso:
Finger weg von der Freispeicherverwaltung!

Initialisierung und Finalisierung von Objekten

Vorkurs C/C++, Olaf Bergmann

Initialisierung von Objekten

- Objekt anlegen und benutzen
  ```
  Point pnt;
  pnt.draw();
  ```
- Wie wird ein Objekt der Klasse `Point` initialisiert?

```
class Point {
protected:
  double _x, _y;
public:
  void draw();
};
```

```
class Point {
protected:
  double _x, _y;
public:
  Point(double x = 0, double y = 0);
  void draw();
};

Point::Point(double x, double y) : _x(x), _y(y) {}
```

```
Point null, pnt(100.5, 53);

null.draw();
pnt.draw();
```

Default-Konstruktoren

- Kein Konstruktor angegeben
  → automatisch erzeugter Default-Konstruktor
- sonst: explizite Definition erforderlich

```
class Point {
protected:
  double _x, _y;
public:
  Point(double x, double y) : _x(x), _y(y) {}
  void draw();
};

Point pnt;                  // Error
Point null(0,0);            // Ok
```

Copy-Konstruktor

- Erzeugt 1:1-Kopie eines Objekts (→ shallow copy)

```
class Point {
protected:
  double _x, _y;
public;
  Point(double x = 0, double y = 0);
  void draw();
};

Point null;
Point p(null);
```

```
p._x = null._x;
p._y = null._y;
```

```
b.items = a.items
→ beide Member zeigen nun
auf dieselbe Speicheradresse!
```

```
class MyArray {
  Object *items;
  size_t count;
public:
  void free_all();
};

MyArray a;
/* a befüllen ... */
MyArray b(a);
a.free_all();
```

Copy-Konstruktor

- Copy-Konstruktor anpassen

```
class MyArray {
  Object *items;
  size_t cnt;
public:
  MyArray() : items(nullptr),
              cnt(0) {}
  MyArray(const MyArray &);
  void free_all();
};

MyArray::MyArray(const MyArray &a) {
 ... items kopieren ...
}

MyArray a;
/* a befüllen ... */
MyArray b(a);
a.free_all();
```

oder verbieten

```
class MyArray {
  Object *items;
  size_t cnt;
public:
  MyArray(); /* ... */
  MyArray(const MyArray &) = delete;
  void free_all();
};



MyArray a;
/* a befüllen ... */

MyArray b(a);    // nicht erlaubt
```

Zuweisungsoperator

```
MyArray a;
MyArray b = a; // Copy-Konstruktor!

b = a;         // operator=
```

```
class MyArray {
  Object *items;
  size_t count;
public:
  MyArray &operator=(const MyArray &);
};

MyArray &MyArray::operator=(const MyArray &rhs) {
  /* ... items kopieren ... */
  return *this;
}
```

Destruktor

```
{
  MyArray a;
  /* ... a befüllen ... */
}
```

a.items gehen verloren!

```
class MyArray {
  Object *items;
  size_t count;
public:
  ~MyArray();
};

MyArray::~MyArray() {
  free_all();
}
```

Move-Semantik

Verschieben statt Kopieren von *rvalues*

```
class Objekt {
  ...
};

Objekt f(void) {
   Objekt k;
   return k;
}

int main() {
   Objekt x = f();
}
```

k ist temporär, kann nicht weiter verwendet werden

x existiert bis zum Ende von main()

x wird mit k initialisiert

Noch mehr Konstruktoren

- Wie weist man `unique_ptr` einander zu?

```
class A {
private:
  string s;
public:
  A() : s("A") {}
  A(A&& a) : s(move(a.s)) {}
  void show() const {
    cout << "A: " << s << endl;
  }
};
```

rvalue-Referenz

```
A a;
A b(move(a));
a.show();
b.show();
```

```
unique_ptr<A> a(new A);

unique_ptr<A> b(move(a));
```

Konstruktoren: default und delete

```
class A {
  int a;
public:
  A(int value) : a(value) {}
  A() = default;
};

...
A a(7);
A b;
```

Default-Konstruktor erzeugen lassen

nicht erlaubt: kein Default-Konstruktor

```
class A {
  Object *items;
public:

  A(const A &) = delete;
};

...
A a;
A b(a);
```

Konstruktor explizit löschen

Copy-Konstruktor wird automatisch erzeugt

Rule of Five

Default-Konstruktoren und Destruktor werden nur automatisch erzeugt, wenn weder Konstruktor noch Destruktor definiert wurden.

- Immer Klassendefinition angeben:
  - nichts
  - Destruktor, Copy-Konstruktor, Copy Assignment Operator
  - Destruktor, Copy-Konstruktor, Copy Assignment Operator, **[Move-Konstruktor], Move Assignment Operator**

Smart Pointer

Vorkurs C/C++, Olaf Bergmann

Pointer und Exceptions

```
void f(void) {
{
   Objekt *p = new Objekt;
   throw "Fire!";
   delete p;
}


int main() {
  try {
    f();
  } catch (...) {
    cerr << "Something went wrong!" << endl;
  }
}
```

In `f()` tritt eine Exception auf
→ Funktion wird sofort verlassen
→ p wird **nicht freigegeben**

Smart Pointer

- Pointer in C/C++ sind fehleranfällig und unflexibel
  → `unique_ptr`
- `unique_ptr<T,Deleter=default_deleter>(&`*Object*`)`
  - Alleiniger Eigentümer von *Object*
  - Zerstört *Object* automatisch, wenn nicht mehr sichtbar
  - Zerstört *Object* automatisch, wenn `unique_ptr` anderen Inhalt bekom

Resource Acquisition is Initialization (RAII)

Smart Pointer: Benutzung

```
#include <iostream>
#include <memory>

...
std::unique_ptr<int> p(new int);

*p = 25;
std::cout << p.get() << ": " << *p << std::endl;


auto q = std::make_unique<int[]>(5);
q[3] = 37;
```

Freigeben eines Pointers

```
{
   unique_ptr<Objekt> p = make_unique<Objekt>();
   throw "Fire!"
}
```

Automatische Freigabe bei Zerstörung des `unique_ptr`-Objekts am Blockende.

```
unique_ptr<Objekt> p = make_unique<Objekt>();
p.reset();
```

`reset()` ruft Destruktor auf und gibt belegten Speicher frei

Wie übergibt man einen unique_ptr?

```
#include <memory>

...
unique_ptr<Objekt> p = make_unique<Objekt>();
```

`make_unique<Typ>(...)`: erzeugt und initialisiert Instanz von Typ, liefert Typ *

Falsch: `unique_ptr<Objekt> q = p;`
→ Copy-Konstruktor explizit verboten

```
unique_ptr<Objekt> q = move(p);
```

`std::move()` erzwingt Nutzung des Move-Konstruktors für Initialisierung, `p` wird `nullptr`

shared_ptr

- `unique_ptr`: Es gibt genau einen Verweis, Zuweisung nur per `move()`
- mehrere Verweise auf das selbe Objekt?
  → `shared_ptr`

„Reference Counting“

```
shared_ptr<Objekt> p = make_shared<Objekt>();
shared_ptr<Objekt> q = p;

p.reset();
q.reset();
```

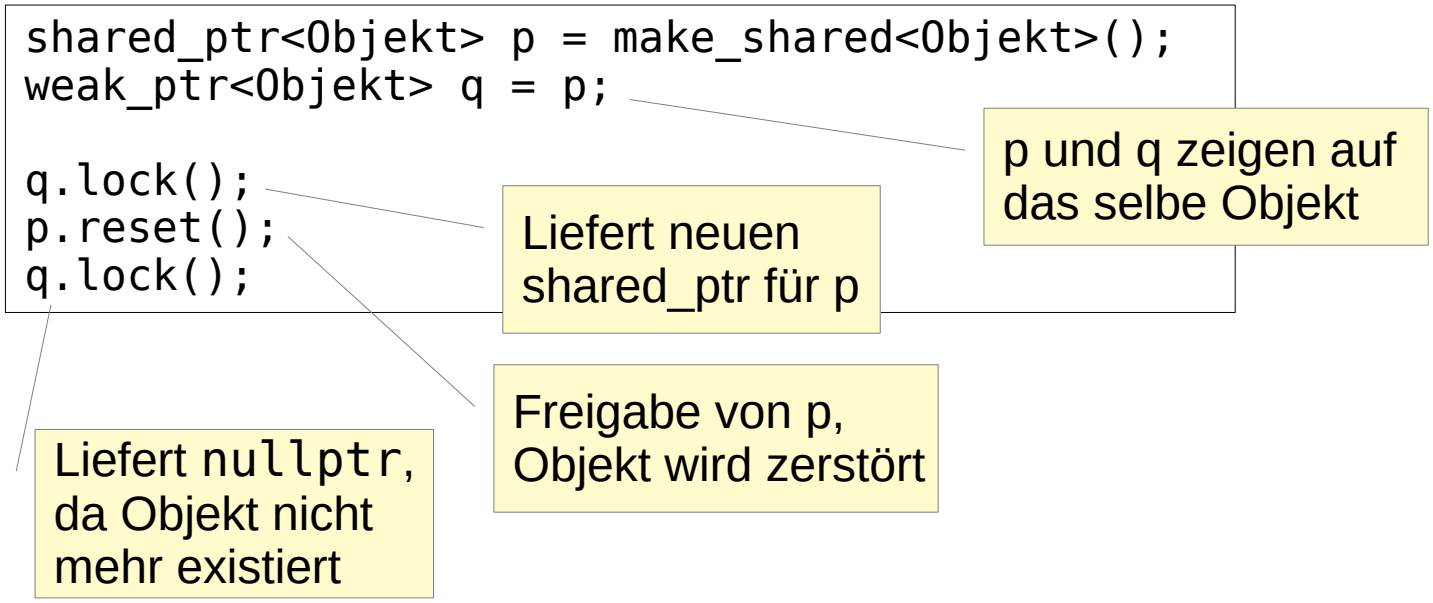
p und q zeigen auf das selbe Objekt

Freigabe von p, q unverändert

Freigabe von q, Objekt wird zerstört

weak_ptr

- `weak_ptr`: sicherer Verweis auf `shared_ptr`
- Reference counter wird nicht erhöht



Generizität mit Templates

Vorkurs C/C++, Olaf Bergmann

Namensräume

- Namenskonflikte vermeiden über `namespace`
  - Schachtelbar
  - Benutzung mit Scope-Operator `::`
- `std`: Namensraum für Definitionen aus der Standard-Bibliothek
- Leerer Bezeichner für oberste Ebene

```
namespace X {
  namespace Y {
    int i;
  }
}

X::Y::i = 7;
```

```
bool
Obj::read(const std::string &f) {
  ::read(file...);
}
```

Argument-dependent Lookup

- Überladene Funktionen werden im Namespace der Argumente gesucht:

```
std::cout << 5;
```

- Idiom:

```
using std::swap;
swap(a, b);
```

Kein operator << im globalen Namespace, aber definiert für std::cout.

Verwendet std::swap, falls kein spezifisches swap auf a oder b definiert

Templates

- Klasse oder Funktion mit Typ-Parameter

```
template <class T>
class vector {
public:
  ...
  int size() const;
private:
  int sz;
  T *p;
};

template<class T>
int vector<T>::size() const { return sz; }
```

Benutzung:

```
vector<int> vi;
vector<char> vc;
```

Typinferenz: decltype

Wdh.:

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i = zahlen.begin(); i != zahlen.end(); ++i) {
  Zahlen::value_type z = *i;
  /* ... */
}
```

viel zu schreiben,
sehr unübersichtlich

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::co                    = za
  decltype(*i) z = *i;
  /* ... */
}
```

... aber immer noch
viel zu schreiben ...

decltype(*expr*):
bestimmt Typ von *expr* zur Übersetzungszeit

```
decltype(2 + 7)     → int

Point *p = nullptr;
decltype(*p)        → Point&

Typ::iterator i;
decltype(*i)        → Typ::value_type&
```

Typinferenz: auto

Wdh.:

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i = zahlen.begin(); i != zahlen.end(); ++i) {
  Zahlen::value_type z = *i;
  /* ... */
}
```

Zahlen::value_type ergibt sich aus *i

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i =
  auto z = *i;
  /* ... */
}
```

auto:
Typ zur zur Übersetzungszeit ermitteln

```
int x, a[10];
auto y = x;       → int

auto p = a;       → int *
auto *q = &a[0]; → int *
auto &r = a[0];  → int &

Typ::iterator i;
auto v = *i;      → Typ::value_type &
```

Typinferenz: auto

Wdh.:

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (Zahlen::const_iterator i = zahlen.begin(); i != zahlen.end(); ++i) {
  auto z = *i;
  /* ... */
}
```

int &

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (int& z : zahlen) {

  /* ... */
}
```

```
typedef vector<int> Zahlen;

Zahlen zahlen = { /* ... */ }

for (auto& z : zahlen) {

  /* ... */
}
```

Typinferenz: auto als Rückgabetyp

```
auto fib(unsigned long i)
{
   switch (i) {
   case 0:
   case 1:
      return i;
   default:
      return fib(i - 2) + fib(i - 1);
   }
}
```

Bestimmt Rückgabetyp

Rekursion erlaubt, wenn Rückgabetyp bekannt

Typinferenz: auto als Rückgabetyp

```
auto fib(unsigned long i) -> decltype(i)
{
   switch (i) {
   case 0:
   case 1:
      return i;
   default:
      return fib(i - 2) + fib(i - 1);
   }
}
```

benötigt, wenn Rückgabetyp nicht automatisch bestimmbar

Bestimmt Rückgabetyp

Rekursion erlaubt, wenn Rückgabetyp bekannt

Algorithmen

```
#include <numeric>
#include <vector>

double
average(const std::vector<double> &v) {
  return std::accumulate(v.cbegin(), v.cend(), 0.0) / v.size();
}
```

Mit Lambda-Ausdruck:

```
std::accumulate(v.cbegin(), v.cend(), 0.0,
     [&] (double init, double val) {
         return init + val / v.size();
     });
}
```

erlaubt Variablen-Referenzen

Algorithmen und Iteratoren

```
#include <random>
#include <vector>
#include <algorithm>

using namespace std;

vector<int> v;
generate_n(inserter(v, v.end()), 100, mt19937());
```

erzeugt Werte für Container

Pseudo-Zufallszahlengenerator

erzeugt einen Insert-Iterator für v

```
#include <iostream>
#include <iterator>

using namespace std;

copy(v.begin(), v.end(), ostream_iterator<int>(cout, ", "));
```

kopiert Elemente von [begin(), end())

Ziel der Kopie: Ausgabeiterator

Generator

```
#include <random>
#include <vector>
#include <algorithm>

using namespace std;

class Data {
  int wert = 0;
public:
  int operator()(void) { return wert++; }
};

vector<int> v;
generate_n(inserter(v, v.end()), 100, Data());
```

spart den Konstruktor für die Initialisierung

Funktionsaufrufoperator, z. B.: Data()();

Basisdatentypen

Vorkurs C/C++, Olaf Bergmann

Grunddatentypen

- Interpretation von Speicher (ia32)

00000000	00000000	00000000	00000001

- Bytes mit jeweils 8 Bit (Bits nicht einzeln adressierbar)
- Längere Datenobjekte (Wörter)
  - Semantik?
  - z. B. Reihenfolge der Bytes in einem Wort (Byte-order)

Arten von Grunddatentypen

- Ganze Zahlen
  - `[signed|unsigned] char`
  - `[signed|unsigned] [short|long[long]] int`
  - `wchar_t`
  - `enum`-Typen
- Gleitkomma-Typen (Floating Point)
  - `float`
  - `[long] double`
- `void`

Speicherbedarf der Grunddatentypen

# Bytes	signed	unsigned
0	void	
1	char	
?	wchar_t	
1	signed char	unsigned char
h ≥ 2	[signed] short [int]	unsigned short [int]
i ≥ h	[signed] int	unsigned [int]
j ≥ i	[signed] long [int]	unsigned long [int]
k ≥ 2	float	—
m ≥ k	double	—
n ≥ m	long double	—

Größen der Datentypen heute

Typ	anzunehmen	ILP32	LP64
char	[≥] 8 Bits	8 Bits	8 Bits
short	≥ 16 Bits	16 Bits	16 Bits
int	short ≤ int ≤ long	32 Bits	32 Bits
long	≥ 32 Bits	32 Bits	64 Bits

- in Bibliotheken definiert
  - `size_t`: unsigned, passend für Größe eines Objekts
  - `ptrdiff_t`: signed, passend für Differenz zwischen Zeigern
  - `int`*N*`_t`, `uint`*N*`_t`: für Variablen fester Größe (optional, $N \in \{ 8, 16, 32, 64 \}$)

<cstdint>

Konstanten

- Integerzahlen

`1234`		(int)
`123456789L`	`0l`	(long)
`1234u`		(unsigned)
`123456789Ul`	`0uL`	(unsigned long)
`0b1000001`	`0b0ULL`	(binär)
`0777`	`0777u`	(oktal)
`0x3F`	`0x3FL`	(hexadezimal)

- Gleitkommazahlen

`1234.5`	`1e-6`	(double)
`1234.5f`	`1e-6F`	(float)
`1234.5L`	`1e-6l`	(long double)

Character-Konstanten

```
'a'     '9'   '\n'    '\t'    '\''          '\0'
'\177'  '\7'  (oktal)    '\xb'   (hexadezimal)
```

Rechnen mit chars:
`'a' + 1 == 'b'`
→ Asciitabelle
(nicht sehr portabel)

Bedeutung	Kürzel	C-Esca
new-line	NL (LF)	\n
horizontal tab	HT	\t
carriage return	CR	\r
alert	BEL	\a
backslash	\	\\
single quote	'	\'
double quote	„	\“
octal number	ooo	\ooo
hex number	hh	\xhh

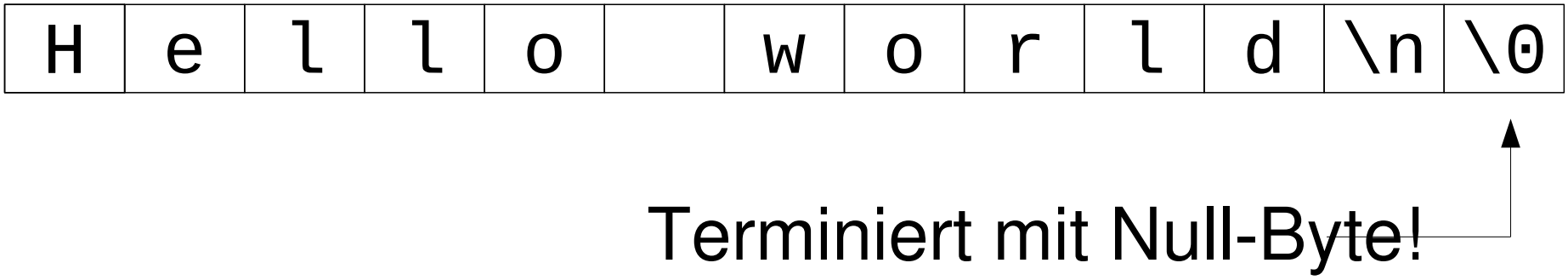
Zeichenketten

```
"Hello world\n"

"Hello"    " world\n"

""
```

Leerstring (ein Null-Byte)



Aufzählungstypen

Vorkurs C/C++, Olaf Bergmann

Read-only Variablen

- Schlüsselwort `const`

```
const double pi = 3.14159265359;
const int off = 0, running = 1;
```

- Enumeration

```
enum { off, idle, running };
```

anstatt

```
const int off = 0;
const int idle = 1;
const int running = 2;
```

Aufzählungstypen 1

```
enum state { off, idle, running };
enum months { jan = 1, feb, mar, ... };
```

muss in `int` passen

Typdefinition

```
enum beer { ale, bitter, lager, stout };
enum juice { orange, apple, grapefruit };
```

Benutzung

```
enum beer carlsberg = lager;
enum juice granini = orange;

granini = lager;
int i = lager;

if (carlsberg == orange) {
   ...
}
```

error

i == 2

error

Aufzählungstypen 2

Typdefinition

```
enum state { off, idle, running };
enum months : uint8_t { jan = 1, feb, mar, ... };

enum class Steuercode : char { ESC = 0x1b };
```

Benutzung

```
state st;
months m;
Steuercode c;

switch (c) {
  case Steuercode::ESC: …
}
```

int, d. h. 32 Bit

uint8_t, d. h. 8 Bit

Operationen und Ausdrücke

Vorkurs C/C++, Olaf Bergmann

Operationen und Ausdrücke

- Operation, Operand
  `3 + 4` (binär)
  `-2` (unär)
- Arithmetische Operatoren
  - Grundrechenarten: `+` `-` `*` `/`
  - Modulo: `%`

schneidet ab zwischen ganzen Zahlen
erzeugt Gleitkomma zwischen Gleitkommazahlen:
`7/2 → 3` `7.0/2 → 3.5`

Relationale und logische Operatoren

- Ergebnis ist wahr (`1`) oder falsch (`0`)
- Relational: `>` `>=` `<` `<=` `==` `!=`
- Logisch: `&&` (und) `||` (oder)
- Auswertung bricht ab, wenn Ergebnis klar

```
while (i < MAXBUF && buf[i] != '\0') {
  i++;
}
```

`0 && X`	`0`
`1 && X`	`X`
`0 || X`	`X`
`1 || X`	`1`

Wäre für i ≥ MAXBUF illegal

- Unäre Negation: `!`

`!!X ≡ X != 0`

`if (!`*expression*`) ≡ if (`*expression*` == 0)`

Bitweise Operatoren

&	bitweises UND
|	bitweises ODER
^	bitweises XOR
<<	Linksshift (füllt mit 0 auf)
>>	Rechtsshift (füllt mit 0 auf)
~	bitweises Komplement (unär)

Achtung!

&& und || ≠ & und |

```
(1 & 2) == 0
(1 && 2) == 1
```

- Bits setzen

```
x = x | BITS          x = x | (1 << BITNUM)
```

- Bits ausmaskieren (löschen)

```
x = x & 0177          x = x & ~0xF
```

Beispiel: gesetzte Bits zählen

```
int bitcount(unsigned x) {
  int count;
  for (count = 0; x != 0; x = x >> 1) {
    if (x & 1) {
      count++;
    }
  }
  return count;
}
```

besser:

```
#include <bitset>

bitset<32> bits("0b001101");
bits.count();
```

Typdefinition und Typumwandlung

Vorkurs C/C++, Olaf Bergmann

typedef: neue Typbezeichner

Typdefinition

```
class X { … };

typedef X MeinX;
typedef map<char, unsigned> Histogram;
```

Benutzung

```
MeinX m;
Histogram h;

h['C'] = 17;
```

typedef: neue Typbezeichner

Typdefinition

```
typedef class {
public:
   …
} Auto;
```

anonyme Klasse

neuer Bezeichner für anonymen Typ

Benutzung

```
Auto pkw;
```

Typumwandlung

- Arithmetische Operationen nur für `int` oder größer definiert
- *integer promotion*
  → Umwandlung der Operanden nach `int`
  (`[[un]signed] char`, `short`)
  → oder nach unsigned int
  (`unsigned short`, `enum`, `wchar_t`)
- Ausdruck mit Operanden unterschiedlichen Typs
  - Automatische Typumwandlung → gemeinsamer Typ, möglichst ohne Informationsverlust
- Umtypungen in arithmetischen Ausdrücken
  - „niedriger“ Typ wird zu „höherem“ konvertiert
  - (ganzzahlig) → `float` → `double` → `long double`
  - `int` → `unsigned` → `long` → `unsigned long`

Noch mehr Typumwandlungen

- Zuweisung
```
int i = 1234;
char c;
c = i;
```
- Funktionsaufruf
```
double sqrt(double);
...
double result = sqrt(2);
```
- cast-Operator
  - Explizite Typumwandlung: `(Typname)` Ausdruck
```
double result = sqrt((double)2);
```

Häufiges Idiom (nicht portabel!):
```
zahl = zahl * 10 + (zeichen – '0');
```

`c → -46`

Typumwandlungen in C++

```
class A {
  int x;
public:
  A(int X) : x(X) {}
  operator int() const { return x; }
};
```

Typecast-Operator

```
A a(1);

if (a) {
  ...
}
```

Implizite Typwandlung nach `bool`

explicit

```
class A {
  int x;
public:
  A(int X) : x(X) {}
  explicit operator int() const { return x; }
};
```

verbietet implizite Wandlung

```
A a(1);

if (a) {
  ...
}
```

Fehler: `operator bool` nicht definiert

explicit

```
class A {
  int x;
public:
  explicit A(int X) : x(X) {}
  explicit operator int() const { return x; }
};
```

verbietet implizite Wandlung

```
A a = 1;

if (a != 0) {
  ...
}
```

Fehler: Copy-Initialisierung erfordert implizite Wandlung nach A(1)

Sichere Typumwandlungen

- Gefährlich in C

```
char a[10];
int *iptr;
iptr = (int *)(a+1);
```

keine weitere Überprüfung
„Der Programmierer hat recht.“

- In C++:

```
iptr = static_cast<int *>(a+1);
```

```
→ /tmp/i.cc:8:32: error: invalid static_cast from type ‘char*’ to type ‘int*’
   iptr = static_cast<int *>(a+1);
```

dynamic_cast

```
class Basis {
public:
  virtual ~Basis();
};

class Abgeleitet : public Basis {
public:
  virtual void f(void);
};

void func(Basis &b) {
  Abgeleitet &ar = b;
  ar.f();
}

Abgeleitet a;
func(a)
```

Fehler!

dynamic_cast

```
class Basis {
public:
  virtual ~Basis();
};

class Abgeleitet : public Basis {
public:
  virtual void f(void);
};

void func(Basis &b) {
  Abgeleitet &ar = dynamic_cast<Abgeleitet&>(b);
  ar.f();
}

Abgeleitet a;
func(a)
```

`throw std::bad_cast`
falls b nicht kompatibel
(`nullptr` für Pointer)

const_cast

```
class Objekt {
public:
  void f(void);
};

void func(const Objekt &ob) {
  ob.f();
}
```

Fehler!

Objekt::f ist nicht const

const_cast

```
class Objekt {
public:
  void f(void);
};

void func(const Objekt &ob) {
  const_cast<Objekt&>(ob).f();
}
```